Exercise 1: Inventory Management System

Ans1- Data structures and algorithms are essential to dealing with large inventories

because:

1 They enable a program to store, retrieve, and manipulate data efficiently, a

task important for large inventories.

2 Poor data structures elevate the time complexity of their operations, affecting

the overall system.

3 The right data structures guarantee that the system scales well when the

number of products increases.

Ans2. Suitable Data Structures

This structure provides facilities of the dynamic array therefore it is useful when speedy index based access is needed.

It provides an efficient storage of a set of key and value pairs. It can be used to look up, add, or delete a product quickly by product ID.

The elements are kept in order, therefore useful when one has to traverse products in sorted order frequently.

Time Complexity Analysis :

A) Add Product: In this case, the time complexity will be O(1) in the average case because of the efficient insertion of key-value pairs in HashMap.

B) Update Product: Here, also in the time complexity, it is O(1) in terms of the average case as it includes key lookup or insertion in the HashMap.

C) Delete Product: In this case also, as it is an efficient removal of a key-value pair from a HashMap, its average case time complexity will be O(1).

D) Fetch product: In this case, the time complexity would be O(1) averaged, as lookups by key are very fast in a HashMap.

Optimization Discussion:

A) Batch Operations: Several products can be added, updated, or deleted in one go, all without being burdened by the overhead of processing each operation individually.

B) Concurrency Handling: In the case of a multi-threaded environment, Concurrent HashMap can replace HashMap to provide thread-safety over the operations.

C) Caching: Often accessed products could be cached for the items of very high popularity.

Exercise2: E-commerce Platform Search Function

Big O Notation

Ans1. Big O notation is the mathematical tool that is used to explain the efficiency of an algorithm. The primary application of Big O lies in the description of the performance of an algorithm, usually expressed as a function of the size of input, typically represented as 'n'. It provides a way of expressing an algorithm's time and space complexity.

A) Best Case: This refers to the case when the algorithm performs the least number of steps.

b) Average Case: This is the typical case; here, the algorithm runs for an average number of steps.

C) Worst Case: This is when the algorithm runs for the maximum number of steps.

Analysis:-

Time Complexity Comparison

Linear Search:

A) Best Case: O(1) Item is at the beginning

b) Average Case: O(n) Item is in the middle.

C) Worst Case: O(n) This would be when the searched-for element is at the end or not found at all.

Binary Search:

A) Best Case: O(1) (Item found at the middle itself)

B) Average Case: O(log n) The array gets halved with every step.

c) Best Case: O(log n). Item will be available at the end of the search.

Suitability for the Platform

A) Linear search is simple to implement and can be done on an unsorted array. It is best for small datasets in which sorting is not worth the extra trouble.

B) Binary Search is much faster on big datasets, but it requires the array to be sorted first. This adds an additional O(n log n) time for sorting.

C) Binary Search would normally be more appropriate for an e-commerce platform since it works a great deal faster in large sets of data. Since this is a search for products, which happens quite often, the early cost of sorting is easily overcome by the subsequent searching speed.

Exercise 3: Sorting Customer Orders

Understand Sorting Algorithms

Bubble Sort:

A. A comparison-based simple algorithm.

B) It repeatedly goes through the list, comparing adjacent elements and swapping them if they are in the wrong order.

c) Best Case: Time complexity O(n), Average and Worst Case: $O(n^2)$.

Insertion Sort:

A) Builds the sorted array one element at a time.

b) All elements greater than the key are moved one position forward from where they are.

c) Time complexity: Best case O(n), average case and worst case $O(n^2)$.

Quick Sort:

A) A divide-and-conquer algorithm.

B) Chooses a pivot element and partitions the array so that, when the pivot is in its final position, the smaller elements are to its left and the larger elements.

C) Time Complexity: Best and Average Case O(n log n); Worst Case $O(n^2)$.

Merge Sort:

A) A divide-and-conquer algorithm.

B) Divides the array into two halves, recursively sorting the two halves, then merges them together.

C) The time complexity is O\(n log n) in all cases.

Analysis

Performance Comparison

Bubble Sort:

Best Case: O(n)

AVERAGE CASE AND WORST CASE : $O(n^2)$

Though easy to implement, it is not efficient for large datasets because it has quadratic time complexity.

Quick Sort:

Best and Average Case: O(n log n)

Worst Case: O(n^2) – very rare, occurring only when the choice of the pivot is poor.

Generally preferred because of its average case performance and efficient utilization of

Memory.

Why Quick Sort is Preferred

Efficiency: On average, and in the best case, Quick Sort has a time complexity of O(n log n), which is better than Bubble Sort.

Memory Usage: Quick Sort is an in-place sorting algorithm; it uses only a small additional amount of fixed storage.

Real-World Performance: In practice, Quick Sort is generally faster as a result of effective cache performance and the divide-and-conquer technique that makes it very suited for parallel processing.

Exercise 4: Employee Management System

Array Representation in Memory

A. Contiguous Memory Allocation: Arrays are stored in contiguous memory locations, where each element is placed side by side of the previous element, which facilitates direct access through indexing.

B. Fixed Size: The size of an array is fixed at the time it is declared and cannot be dynamically changed.

C. Efficient Access: It has access to elements efficiently in O(1) time complexity due to the predictability in memory structure.

Advantages of Arrays

A) Constant Time Access: One can get an element very fast using its index by direct indexing.

B) Efficient Use of Memory: Arrays use a single contiguous block of memory, thus reducing overhead.

C) Cache Performance: Spatial locality coming from the contiguous memory allocation of arrays improves cache performance.

Analysis

Time Complexity Analysis

Add Employee: O(1) - This is because addition of an employee at the end of the array takes only constant time in the worst case when sufficient space is available.

Search Employee: O(n) - In the worst scenarios, all array employees will have to be covered in a search.

Traverse Employees: O(n) - It becomes a simple matter of visiting each and every employee in the array.

Delete Employee: O(n) - In the worst case, the deletion of an employee requires that elements be shifted to fill up the gap left.

Limitations of Arrays

Fixed Size : Since arrays are of a fixed size, it may lead to either inefficient memory use, or it would not be possible to store more elements than the array was created to hold.

Insertion and Deletion : Inserting or deleting elements is inefficient as shifting of rest elements is needed in order to facilitate space for the new addition or deletion.

Sparse Data : It does not support sparse data structures, in which the elements are widely spread within the array range.

When to Use Arrays

Fixed Size Collection: Good when you've got a foreknowledge of how much nuances there will be, and doesn't change.

Fast Access: Allows for instant element finding by its index.

Memory Contiguity: You do not need anything else if you want to conserve the memory that other structures could waste, whilst requiring contiguous memory allocation.

Exercise 5: Task Management System

Understand Linked Lists

Types of Linked Lists

Singly Linked List:

- Each node contains data and a pointer that refers to the next node in the sequence.

- Operations are easily performed, but can be traversed in one direction only.

Doubly Linked List:

- Node Structure: These nodes have data and a reference to the next node and one to the previous node.

- Bidirectional Traversal: The program may traverse it forward and backward.

- Memory Use: It consumes more memory because an additional reference is required.

Time Complexity Analysis:

- Add Task: O(n) — In the worst case, to add a task, it needs to travel to the end of the list.

- Search Task: O(n) - Worst case, all elements in the list may need to be visited to search for a task.

- Traverse Tasks: O(n) - Visiting every single element in the list.

- Delete Task: O(n) - In the worst case, deleting a task may require that every element in the list be visited to search for the task.

Advantages of Linked Lists Over Arrays:

- Dynamic Size: A linked list has the advantage of dynamic size; it can grow or shrink as per requirements, while an array has a fixed size.

- Efficient Insertions/Deletions: Operations such as insertions and deletions are performed more efficiently, especially in lists of large size or when end-user operations are done near the beginning of the list.

- Memory Utilization: In scenarios where the number of elements is unknown or varies a lot, then a linked list may be more memory efficient as it need not use up a contiguous block of memory for its fixed size.

Limitations of Linked Lists:

-Access time: In a Linked List, the access time of its elements is O(n) because one has to traverse sequentially; for arrays it's O(1).

-Memory Overhead: A linked list requires additional memory for each node to store the reference to the next node, which can be huge for large lists.

Exercise 6: Library Management System

Linear Search Algorithm:

- **Description:** Linear search goes through every element of the list linearly till it gets a match or the end of the list.

- **Time Complexity:**

  - Best Case: O(1) in case of beginning placement

  - Average Case: O(n)

  - Worst Case: O(n) if the element appears at the last or not at all

Binary Search Algorithm:

Description: Binary search is a technique applied to a sorted list; the technique repeatedly divides the search interval in half. It requires comparing the target value with the middle element to decide which half to proceed searching for.

Time Complexity:

  - Best Case: O(1) (provided that the middle element is the target)

  - Average Case: O(log n)

  - Worst Case: O(log n)

Time Complexity Comparison:

Linear Search:

- Best Case: O(1)

- Average Case: O(n)

- Worst Case: O(n)

Binary Search:

- Best Case: O(1)

- Average Case: O(log n)

- Worst Case: O(log n)

When to Use Each Algorithm:

- Linear Search:

- To be used on unsorted lists.
- It should be used on very small datasets for which sorting isn't worth the trouble.
- It's simple and requires no setup overhead.

- Binary Search:

- Applied in sorted lists.
- It is more efficient for larger datasets since it has a logarithmic time complexity.

Exercise 7: Financial Forecasting

Definition: It is a design technique that solves a problem by breaking it into smaller instances of related problems, in which a function calls itself as a subroutine. In other words, during execution, a function calls itself to repeat the process several times.

Base and Recursive Case: The recursive function has to contain a base case that stops the recursion, and a recursive case that expresses the problem in a simpler form.

It often makes it easier to implement solutions that involve complex problems. The reason is that it breaks down complex problems into simpler, more manageable sub-problems.

Complexity Analysis:

Time Complexity: The time complexity of this recursive algorithm is O(n), where 'n' is the number of periods. Every call to the `predict Future Value` method decrements the number of periods by 1, which is a total of 'n' recursive calls.

Memoizing the Recursive Solution:

Memoization: This method makes a recursive solution optimum. Store the results of expensive function calls and return the cached result when the same inputs occur again.

Iterative Solution: Another way would be to change the recursive solution into its iterative version. It might turn out to be more memory-efficient and helps to avoid the risk of stack overflow.